|  |  |
| --- | --- |
| C:\user-pjpf\Briefcase-SO\so\so-12-13\slides\logo-IST-Tecnico.JPG | INSTITUTO SUPERIOR TÉCNICO  Departamento de Engenharia Informática  MEIC 2021-2022 – 3rd Period  Sistemas de Elevada Confiabilidade |

Project Report – Group 4

**Nuno Estalagem 102245  
Francisco Cabrinha 102141  
Ricardo Ribeiro 102108**

# Design, Threat Analysis and Protection Mechanisms

Our BFTB system was implemented in Java. Each client communicates with the Servers via GRPC channels, and each Server persists data/information of each account atomically (i.e. either completes or has no effect), thus being corruption prevented.

We divided our system in 2 modules:

* A Server whose responsibility is answering to Client requests
* A User which is the Client of the system.

Both modules contain a Class responsible for performing all the Digital Signatures/Digital Signature verifications made throughout the System. Additionally, both modules contain an API that allow them to obtain each Server’s Public Key and their own key pair.

* 1. **Project Layers**

Uma imagem com mesa

Descrição gerada automaticamente

Picture 1- BFTB Client System

Uma imagem com mesa

Descrição gerada automaticamente

Picture 2- BFTB Server System

* 1. **System Considerations**
     1. **Base considerations**

To tolerate *f* byzantine Servers, our System creates a N number of Servers such that N=3*f*+1

Whenever a Server throws an exception, that exception is signed with the Server’s private Key together with the sent RID/WID to prevent man in the middle/ spoofing.

Each Private Key is confidential because it was stored in a Keytool generated Keystore i.e., a repository of security certificates whose content is a Public Key Certificate and its corresponding Private Key.

* + 1. **(1, N) Byzantine Regular Register**

Whenever a **CORRECT** Client sends a request to all Servers, he waits for a Byzantine Quorum of responses, i.e. (N+F)/2+1, meaning that if a transaction has been registered, the last written value will be returned to the Client by at least one Server (assuming that we are tolerating f faults, of course).

* + 1. **(1, N) Byzantine Atomic Register**

Whenever a **CORRECT** Client sends a **read** request to all Servers and after he verifies that he has been sent a valid response by a Quorum of servers, he sends each server a write-back request with the parameters received from his **read** response, here the servers perform ADEB.

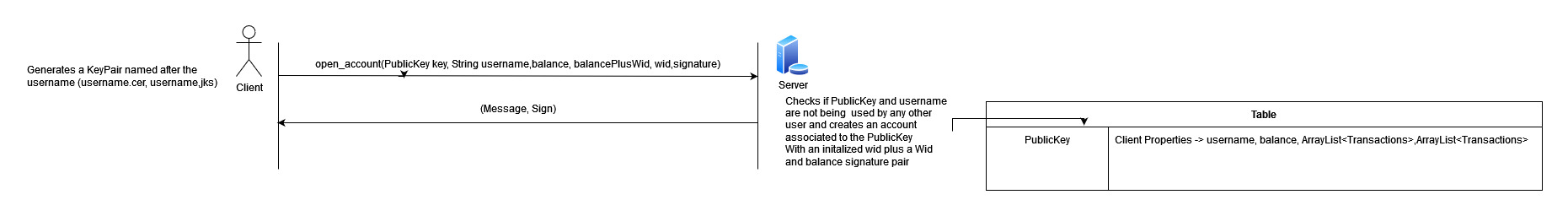
* + 1. **DDOS combat mechanism**

To avoid a **Denial-of-Service** attack where a byzantine client/attacker could flood the server with requests making it unavailable, we implemented a Proof of Work mechanism based on already existing work from some Crypto Currencies like Ethereum and Bitcoin. For example, in Bitcoin, when mining a block, you have to discover a nonce, mix it with a block, which results in a hash that needs to have a certain number of zeros in the most left values, this is the difficulty, and send it to the system to be verified. Our system the User asks each server for a nonce/challenge mixes it with its input, and generates the hash, with a predetermined difficulty and sends the result to the server. The protected requests were:

* **Check\_account and Audit-** because all servers need to not only search for a user’s pending/finished transactions, but also perform a write back, which must execute ADEB, exchanging a quadratic number of messages between them.
  + 1. **ADEB**

Our system uses ADEB to prevent servers from becoming inconsistent between each other. This algorithm is executed before every write and write back function to ensure that all Servers register the same transactions. More specifically, it is performed to ensure that all Servers stay consistent even if a user sends a different input to different servers (E.g performs a send amount, in which the amount changes in every single server).

* 1. **System API**



Picture 3 - open\_account

**Uma imagem com texto

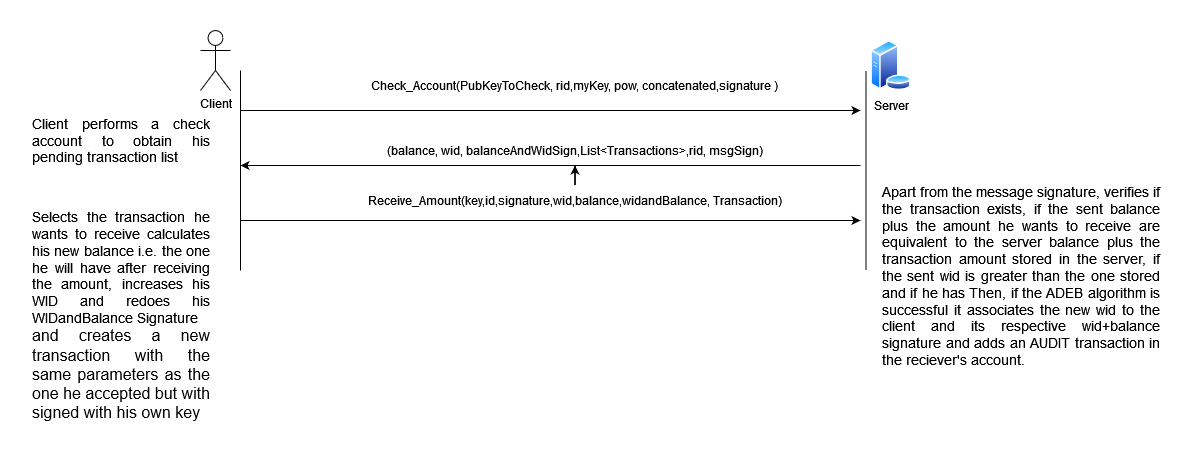
Descrição gerada automaticamente**

Picture 4 - send\_amount

**Uma imagem com mesa

Descrição gerada automaticamente**

Picture 5 - check\_account



Picture 6 - receive\_amount

**Uma imagem com mesa

Descrição gerada automaticamente**

Picture 7 - audit

* 1. **Threat Analysis, Protection Mechanisms and Guarantees**

**(BTFB server -> S, Client -> C and Man in the Middle -> M)**

**T: Send Amount impersonation** -> C2 impersonates C1 and creates transactions from C1’s account using his Public Key, as it is how the server identifies users.

**P:** When generating a transaction, a C1 updates his balance by reducing it with the amount he wants to send, increments his Wid and signs a Wid and balance pair with his private key, which is sent together with the request. C2 is not able to forge a Wid+balance private key signature.

**T:** **Send Amount Replay attack** -> C1 wants to send an amount to C2. M gets the package sent from C1 to S and keeps replaying it, making unintentional transaction from C1 to C2.

**P:** Impossible because C1 increments his Wid on each transaction, meaning the pair Wid+Balance would change.

**T:** **Receive amount impersonation** -> C2 asks S to receive an amount in name of C1 meaning that C1’s account would change its state without his permission.

**P:** C1 also creates a new transaction with the same parameters of the one he wants to receive, except that this time he signs the new parameters with his Private key, also sending the server his new balance, new wid and balance signature and new Wid, meaning that C2 would have to forge C1’s private key to impersonate him.

**T:** **Account resetting** -> C tries to “recreate” an account with his Public Key, resetting his balance

**P** It’s verified if that key is already in use.

**T:** **Transaction Multiple Acceptance** -> Attacker tries to receive multiple times the same transaction.

**P:** The server keeps track of the list of pending transactions**.**

**T:** **Man in the Middle value change** -> M changes the contents of the requests or responses by C and S respectively, like altering the response of the request check/audit account, providing this way altered bank information to the client.

**P:** Sensitive information is hashed for integrity verification. In read requests C sends its Rid, increments it and only accepts responses whose signature includes the same rid value.

**T: Check account impersonation ->** C2 impersonates C1 in check account/ Audit requests and infinitely increases its Rid to block C1’s read requests.

**P:** Read methods now request a Private key Signature of its parameters.

**T: POW Challenge Pre-Computing** an Attacker pre-computes challenges instead of using the nonce provided by the server.

**P:** A user cannot compute the nonce itself as the server associates the nonce it sent to the user who requested it.

**T: Byzantine Users ->** A Client sends different inputs to each server when sending an amount or receiving an amount.

**P:** Transactions are only persisted after performing ADEB.

**T: Byzantine Servers ->** A server creates fake transactions or changes the user’s balance.

**P:** The client creates the transaction now, sending it to the server to confirm that it is correct and, in case it is, it persists the transaction and changes the user’s balance, persisting the sent Wid+balance signature, Wid, balance and transaction. When a user checks his Audit/account he would check every transaction to see if he signed it i.e., to validate it. A server cannot forge a transaction because it does not have a user’s private Key.

**T: Byzantine Server replay transaction attack ->** server replays a user’s transaction.

**P:** When a client creates a transaction, he associates the Wid he is writing the transaction with to the transaction, meaning each transaction will have a different signature. A client will discard a server response and declare it as byzantine if he gets the same signatures